Sandy2x: New Curve25519 Speed Records

Tung Chou

Department of Mathematics and Computer Science
Technische Universiteit Eindhoven, P.O. Box 513, 5600 MB Eindhoven, the Netherlands
blueprint@crypto.tw

Abstract. This paper sets speed records on well-known Intel chips for the Curve25519 elliptic-curve Diffie-Hellman scheme and the Ed25519 digital signature scheme. In particular, it takes only 159 128 Sandy Bridge cycles or 156 995 Ivy Bridge cycles to compute a Diffie-Hellman shared secret, while the previous records are 194 036 Sandy Bridge cycles or 182 708 Ivy Bridge cycles.

There have been many papers analyzing elliptic-curve speeds on Intel chips, and they all use Intel's serial $64 \times 64 \rightarrow 128$-bit multiplier for field arithmetic. These papers have ignored the 2-way vectorized $32 \times 32 \rightarrow 64$-bit multiplier on Sandy Bridge and Ivy Bridge: it seems obvious that the serial multiplier is faster. However, this paper uses the vectorized multiplier. This is the first speed record set for elliptic-curve cryptography using a vectorized multiplier on Sandy Bridge and Ivy Bridge. Our work suggests that the vectorized multiplier might be a better choice for elliptic-curve computation, or even other types of computation that involve prime-field arithmetic, even in the case where the computation does not exhibit very nice internal parallelism.

Key words: Elliptic curves · Diffie-Hellman · signatures · speed · constant time · Curve25519 · Ed25519 · vectorization

1 Introduction

In 2006, Bernstein proposed Curve25519, which uses a fast Montgomery curve for Diffie-Hellman (DH) key exchange. In 2011, Bernstein, Duif, Schwabe, Lange and Yang proposed the Ed25519 digital signature scheme, which uses a fast twisted Edwards curve that is birationally equivalent to the same Montgomery curve. Both schemes feature a conservative 128-bit security level, very small key sizes, and consistent fast speeds on various CPUs (cf. [1], [8]), as well as microprocessors such as ARM ([3], [16]), Cell ([2]), etc.

Curve25519 and Ed25519 have gained public acceptance and are used in many applications. The IANIX site [17] has lists for Curve25519 and Ed25519 deployment, which include the Tor anonymity network, the QUIC transport layer network protocol developed by Google, openSSH, and many more.

This paper presents `Sandy2x`, a new software which sets speed records for Curve25519 and Ed25519 on the Intel Sandy Bridge and Ivy Bridge microarchitectures. Previous softwares set speed records for these CPUs using the serial multiplier. `Sandy2x`, instead, uses of a vectorized multiplier. Our results show that previous elliptic-curve cryptography (ECC) papers using the serial multiplier might have made a suboptimal choice.

A part of our software (the code for Curve25519 shared-secret computation) has been submitted to the SUPERCOP benchmarking toolkit, but the speeds have not been included in the eBACS [8] site yet. We plan to submit the whole software soon for public use.

This work was supported the Netherlands Organisation for Scientific Research (NWO) under grant 639.073.005.

1.1 Serial Multipliers versus Vectorized Multipliers

Prime field elements are usually represented as big integers in softwares. The integers are usually divided into several small chunks called *limbs*, so that field operations can be carried out as sequences of operations on limbs. Algorithms involving field arithmetic are usually bottlenecked by multiplications, which are composed of limb multiplications. On Intel CPUs, each core has a powerful $64 \times 64 \rightarrow 128$-bit serial multiplier, which is convenient for limb multiplications. There have been many ECC papers that use the serial multiplier for field arithmetic. For example, [1] uses the serial multipliers on Nehalem/Westmere; [6] uses the serial multipliers on Sandy Bridge; [5] uses the serial multipliers on Ivy Bridge.

On some other chips, it is better to use a vectorized multiplier. The Cell Broadband Engine has 7 Synergistic Processor Units (SPUs) which are specialized for vectorized instructions; the primary processor has no chance to compete with them. ARM has a 2-way vectorized $32 \times 32 \rightarrow 64$-bit multiplier, which is clearly stronger than the $32 \times 32 \rightarrow 64$ serial multiplier. A few ECC papers exploit the vectorized multipliers, including [3] for ARM and [2] for Cell. In 2014, there is finally one effort for using a vectorized multiplier on Intel chips, namely [4]. The paper uses vectorized multipliers to carry out hyperelliptic-curve cryptography (HECC) formulas that provide a natural 4-way parallelism. ECC formulas do not exhibit such nice internal parallelism, so vectorization is expected to induce much more overhead than HECC.

Our speed records rely on using a 2-way vectorized multipliers on Sandy Bridge and Ivy Bridge. The vectorized multiplier carries out only a pair of $32 \times 32 \rightarrow 64$-bit multiplication in one instruction, which does not seem to have any chance to compete with the $64 \times 64 \rightarrow 128$-bit serial multiplier, which is used to set speed records in previous Curve25519/Ed25519 implementations. In this paper we investigate how serial multipliers and vectorized multipliers work (Section 2), and give arguments on why the vectorized multiplier can compete.

Our work is similar to [4] in the sense that we both use vectorized multipliers on recent Intel microarchitectures. The difference is that our algorithm does not have very nice internal parallelism, especially for verification. Our work is also similar to [3] in the sense that the vectorized multipliers have the same input and output size. We stress that the low-level optimization required on ARM is different to Sandy/Ivy Bridge, and it is certainly harder to beat the serial multiplier on Sandy/Ivy Bridge.

1.2 Performance Results

The performance results for our software are summarized in Table 1, along with the results for [1] and [7]. [1] is chosen because it holds the speed records on the eBACS site for publicly verifiable benchmarks [8]; [7] is chosen because it is the fastest constant-time public implementation for Ed25519 (and Curve25519 public-key generation) to our knowledge. The speeds of our software (as [1] and [7]) are fully protected against simple timing attacks, cache-timing attacks, branch-prediction attacks, etc.: all load addresses, all store addresses, and all branch conditions are public.

For comparison, Longa reported $\approx$ 298 000 Sandy Bridge cycles for the "ECDHE" operation, which is essentially 1 public-key generation plus 1 secret-key computation, using Microsoft's 256-bit NUMS curve [19]. OpenSSL 1.0.2, after heavy optimization work from Intel, compute a NIST P-256 scalar multiplication in 311 434 Sandy Bridge cycles or 277 994 Ivy Bridge cycles.

For Curve25519 public-key generation, [7] and our implementation gain much better results than [1] by performing the fixed-base scalar multiplications on the twisted Edwards

	SB cycles	IB cycles	table size	reference	implementation
Curve25519 public-key generation	54 346	52 169	30720 + 0	**(new) this paper**	
	61 828	57 612	24576 + 0	[7]	
	194 165	182 876	0 + 0	[1] CHES 2011	`amd64-51`
Curve25519 shared secret computation	156 995	159 128	0 + 0	**(new) this paper**	
	194 036	182 708	0 + 0	[1] CHES 2011	`amd64-51`
Ed25519 public-key generation	57 164	54 901	30720 + 0	**(new) this paper**	
	63 712	59 332	24576 + 0	[7]	
	64 015	61 099	30720 + 0	[1] CHES 2011	`amd64-51-30k`
Ed25519 sign	63 526	59 949	30720 + 0	**(new) this paper**	
	67 692	62 624	24576 + 0	[7]	
	72 444	67 284	30720 + 0	[1] CHES 2011	`amd64-51-30k`
Ed25519 verification	205 741	198 406	10240 + 1920	**(new) this paper**	
	227 628	204 376	5120 + 960	[7]	
	222 564	209 060	5120 + 960	[1] CHES 2011	`amd64-51-30k`

Table 1. Performance results for Curve25519 and Ed25519 of this paper, the CHES 2011 paper [1], and the implementation by Andrew Moon "floodyberry" [7]. All implementations are benchmarked on the Sandy Bridge machine "`h6sandy`" and the Ivy Bridge machine"`h9ivy`" (abbreviated as SB and IB in the table), of which the details can be found on the eBACS website [8]. Each cycle count listed is the measurement result of running the software on one CPU core, with Turbo Boost disabled. The table sizes (in bytes) are given in two parts: read-only memory size + writable memory size.

curve used in Ed25519 instead of the Montgomery curve; see Section 3.2. Our implementation strategy for Ed25519 public-key generation and signing is the same as Curve25519 public-key generation. Also see Section 3.1 for Curve25519 shared-secret computation, and Section 4 for Ed25519 verification.

We also include the tables sizes of [1], [7] and `Sandy2x` in Table 1. Note that our current code uses the same window sizes as [1] and [7] but larger tables for Ed25519 verification. This is because we use a data format that is not compact but more convenient for vectorization. Also note that [1] has two implementations for Ed25519: `amd64-51-30k` and `amd64-64-24k`. The tables sizes for `amd64-64-24k` are 20% smaller than those of `amd64-51-30k`, but the speed records on eBACS are set by `amd64-51-30k`.

1.3 Other Fast Diffie-Hellman and Signature Schemes

On the eBACS website [8] there are a few DH schemes that achieve fewer Sandy/Ivy Bridge cycles for shared-secret computation than our software: `gls254prot` from [12] uses a GLS curve over a binary field; `gls254` is a non-constant-time version of `gls254prot`; `kummer` from [4] is a HECC scheme; `kumfp127g` from [13] implements the same scheme as [4] but uses an obsolete approach to perform scalar multiplication on hyperelliptic curves as explained in [4].

GLS curves are patented, making them much less attractive for deployment, and papers such as [14] and [15] make binary-field ECC less confidence-inspiring. There are algorithms that are better than the Rho method for high-genus curves; see, for example, [20]. Compared to these schemes, Curve25519, using an elliptic curve over a prime field, seems to be a more conservative (and patent-free) choice for deployment.

The eBACS website also lists some signature schemes which achieve better signing and/or verification speeds than our work. Compared to these schemes, Ed25519 has the smallest public-key size (32 bytes), fast signing speed (superseded only by *multivariate* schemes with much larger key sizes), reasonably fast verification speed (can be much better if batched verification is considered, as shown in [1]), and a high security level (128-bit).

2 Arithmetic in $\mathbb{F}_{2^{255}-19}$

A radix-2^r representation represents an element f in a b-bit prime field as $(f_0, f_1, \ldots, f_{\lceil b/r \rceil - 1})$, such that

$$f = \sum_{i=0}^{\lceil b/r \rceil - 1} f_i 2^{\lceil ir \rceil}.$$

This is called a radix-2^r representation. Field arithmetic can then be carried out using operations on limbs; as a trivial example, a field addition can be carried out by adding corresponding limbs of the operands.

Since the choice of radix is often platform-dependent, several radices have been used in existing software implementations of Curve25519 and Ed25519. This section describes and compares the radix-2^{51} representation (used by [1]) with the radix-$2^{25.5}$ representation (used by [3] and this paper), and explains how a small-radix implementation can beat a large-radix one on Sandy Bridge and Ivy Bridge, even though the vectorized multiplier seems to be slower. The radix-2^{64} representation by [1] appears to be slower than the radix-2^{51} representation for Curve25519 shared-secret computation, so only the latter is discussed in this section.

2.1 The Radix-2^{51} Representation

[1] represents an integer f modulo $2^{255} - 19$ as

$$f_0 + 2^{51} f_1 + 2^{102} f_2 + 2^{153} f_3 + 2^{204} f_4$$

As the result, the product of $f_0 + 2^{51} f_1 + 2^{102} f_2 + 2^{153} f_3 + 2^{204} f_4$ and $g_0 + 2^{51} g_1 + 2^{102} g_2 + 2^{153} g_3 + 2^{204} g_4$ is $h_0 + 2^{51} h_1 + 2^{102} h_2 + 2^{153} h_3 + 2^{204} h_4$ modulo $2^{255} - 19$ where

$$\begin{aligned}
h_0 &= f_0 g_0 + 19 f_1 g_4 + 19 f_2 g_3 + 19 f_3 g_2 + 19 f_4 g_1, \\
h_1 &= f_0 g_1 + f_1 g_0 + 19 f_2 g_4 + 19 f_3 g_3 + 19 f_4 g_2, \\
h_2 &= f_0 g_2 + f_1 g_1 + f_2 g_0 + 19 f_3 g_4 + 19 f_4 g_3, \\
h_3 &= f_0 g_3 + f_1 g_2 + f_2 g_1 + f_3 g_0 + 19 f_4 g_4, \\
h_4 &= f_0 g_4 + f_1 g_3 + f_2 g_2 + f_3 g_1 + f_4 g_0.
\end{aligned}$$

One can replace g by f to derive similar equations for squaring.

The radix-2^{51} representation is designed to fit the $64 \times 64 \to 128$-bit serial multiplier, which can be accessed using the `mul` instruction. The usage of the `mul` is as follows: given a 64-bit integer (either in memory or a register) as operand, the instruction computes the 128-bit product of the integer and `rax`, and stores the higher 64 bits of in `rdx` and lower 64 bits in `rax`.

The field multiplication function begins with computing $f_0 g_0, f_0 g_1, \ldots, f_0 g_4$. For each g_j, f_0 is first loaded into `rax`, and then a `mul` instruction is used to compute the product; some `mov` instructions are required to move the `rdx` and `rax` to the registers where h_j is stored. Each

monomial involving f_i where $i > 0$ also takes a `mul` instruction, and an addition (`add`) and an addition with carry (`adc`) are required to accumulate the result into h_k. Multiplications by 19 can be handled by the `imul` instruction. In total, it takes 25 `mul`, 4 `imul`, 20 `add`, and 20 `adc` instructions to compute $h_0, h_1, \ldots, h_4$ [1]. Note that some *carries* are required to bring the h_k back to around 51 bits. We denote such a radix-51 field multiplication including carries as $\mathbf{m}$; $\mathbf{m}^-$ represents $\mathbf{m}$ without carries.

2.2 The Radix-$2^{25.5}$ Representation

[3] represents an integer f modulo $2^{255} - 19$ as

$$f_0 + 2^{26} f_1 + 2^{51} f_2 + 2^{77} f_3 + 2^{102} f_4 + 2^{128} f_5 + 2^{153} f_6 + 2^{179} f_7 + 2^{204} f_8 + 2^{230} f_9.$$

As the result, the product of $f_0 + 2^{26} f_1 + 2^{51} f_2 + \cdots$ and $g_0 + 2^{26} g_1 + 2^{51} g_2 + \cdots$ is $h_0 + 2^{26} h_1 + 2^{51} h_2 + \cdots$ modulo $2^{255} - 19$ where

$$\begin{aligned}
h_0 &= f_0g_0 + 38f_1g_9 + 19f_2g_8 + 38f_3g_7 + 19f_4g_6 + 38f_5g_5 + 19f_6g_4 + 38f_7g_3 + 19f_8g_2 + 38f_9g_1,\\
h_1 &= f_0g_1 + f_1g_0 + 19f_2g_9 + 19f_3g_8 + 19f_4g_7 + 19f_5g_6 + 19f_6g_5 + 19f_7g_4 + 19f_8g_3 + 19f_9g_2,\\
h_2 &= f_0g_2 + 2f_1g_1 + f_2g_0 + 38f_3g_9 + 19f_4g_8 + 38f_5g_7 + 19f_6g_6 + 38f_7g_5 + 19f_8g_4 + 38f_9g_3,\\
h_3 &= f_0g_3 + f_1g_2 + f_2g_1 + f_3g_0 + 19f_4g_9 + 19f_5g_8 + 19f_6g_7 + 19f_7g_6 + 19f_8g_5 + 19f_9g_4,\\
h_4 &= f_0g_4 + 2f_1g_3 + f_2g_2 + 2f_3g_1 + f_4g_0 + 38f_5g_9 + 19f_6g_8 + 38f_7g_7 + 19f_8g_6 + 38f_9g_5,\\
h_5 &= f_0g_5 + f_1g_4 + f_2g_3 + f_3g_2 + f_4g_1 + f_5g_0 + 19f_6g_9 + 19f_7g_8 + 19f_8g_7 + 19f_9g_6,\\
h_6 &= f_0g_6 + 2f_1g_5 + f_2g_4 + 2f_3g_3 + f_4g_2 + 2f_5g_1 + f_6g_0 + 38f_7g_9 + 19f_8g_8 + 38f_9g_7,\\
h_7 &= f_0g_7 + f_1g_6 + f_2g_5 + f_3g_4 + f_4g_3 + f_5g_2 + f_6g_1 + f_7g_0 + 19f_8g_9 + 19f_9g_8,\\
h_8 &= f_0g_8 + 2f_1g_7 + f_2g_6 + 2f_3g_5 + f_4g_4 + 2f_5g_3 + f_6g_2 + 2f_7g_1 + f_8g_0 + 38f_9g_9,\\
h_9 &= f_0g_9 + f_1g_8 + f_2g_7 + f_3g_6 + f_4g_5 + f_5g_4 + f_6g_3 + f_7g_2 + f_8g_1 + f_9g_0.
\end{aligned}$$

One can replace g by the f to derive similar equations for squaring.

The representation is designed to fit the vector multiplier on Cortex-A8, which performs a pair of $32 \times 32 \to 64$-bit multiplications in one instruction. On Sandy Bridge and Ivy Bridge a similar vectorized multiplier can be accessed using the `vpmuludq` [2] instruction. The AT&T syntax of the `vpmuludq` instruction is as follows:

```
vpmuludq src2, src1, dest
```

where `src1` and `dest` are 128-bit registers, and `src2` can be either a 128-bit register or (the address of) an aligned 32-byte memory block. The instruction multiplies the lower 32 bits of the lower 64-bit words of `src1` and `src2`, multiplies the lower 32 bits of the higher 64-bit words of`src1` and `src2`, and stores the 64 bits products in 64-bit words of `dest`.

To compute $h = fg$ and $h' = f'g'$ at the same time, we follow the strategy of [3] but replace the vectorized addition and multiplication instructions by corresponding ones on Sandy/Ivy Bridge. Given $(f_0, f'_0), \ldots (f_9, f'_9)$ and $(g_0, g'_0), \ldots (g_9, g'_9)$, first prepare 9 vectors $(19g_1, 19g'_1), \ldots, (19g_9, 19g'_9)$ with 10 `vpmuludq` instructions and $(2f_1, 2f'_1)$, $(2f_3, 2f'_3)$, $\ldots$, $(2f_9, 2f'_9)$ with 5 vectorized addition instructions `vpaddq`. Note that the reason to use `vpaddq`

[1] [1] uses one more `imul`; perhaps this is for reducing memory access.

[2] The starting 'v' indicate that the instruction is the VEX extension of the `pmuludq` instruction. The benefit of using `vpmuludq` is that it is a 3-operand instruction. In this paper we show vector instructions in their VEX extension form, even though vector instructions are sometimes used without the VEX extension.

instead of `vpmuludq` is to balance the loads of different execution units on the CPU core; see analysis in Section 2.3. Each $(f_0g_j, f'_0g'_j)$ then takes 1 `vpmuludq`, while each $(f_ig_j, f'_ig'_j)$ where $i > 0$ takes 1 `vpmuludq` and 1 `vpaddq`. In total, it takes 109 `vpmuludq` and 95 `vpaddq` to compute $(h_0, h'_0), (h_1, h'_1), \ldots, (h_9, h'_9)$. We denote such a vector of two field multiplications as $\mathbf{M^2}$, including the carries that bring h_k (and also h'_k) back to $26 - (k \bmod 2)$ bits; $\mathbf{M^{2-}}$ represents $\mathbf{M^2}$ without carries. Similarly, we use $\mathbf{S^2}$ and $\mathbf{S^{2-}}$ for squarings.

We perform a carry from h_k to h_{k+1} (the indices work modulo 10), which is denoted by $h_k \to h_{k+1}$, in 3 steps:

- Perform a logical right shift for the 64-bit words in h_k using a `vpsrlq` instruction. The shift amount is $26 - (k \bmod 2)$.
- Add the result of the first step into h_{k+1} using a `vpaddq` instruction.
- Mask out the most significant $38 + (k \bmod 2)$ bits of h_k using a `vpand` instruction.

For $h_9 \to h_0$ the result of the shift has to be multiplied by 19 before being added to h_0. Note that the usage of `vpsrlq` suggests that we are using *unsigned* limbs; there is no vectorized arithmetic shift instruction on Sandy Bridge and Ivy Bridge.

To reduce number of bits in all of $h_0, h_1, \ldots, h_9$, the simplest way is to perform the carry chain

$$h_0 \to h_1 \to h_2 \to h_3 \to h_4 \to h_5 \to h_6 \to h_7 \to h_8 \to h_9 \to h_0 \to h_1.$$

The problem of the simple carry chain is that it suffers severely from the instruction latencies. To mitigate the problem, we instead interleave the 2 carry chains

$$h_0 \to h_1 \to h_2 \to h_3 \to h_4 \to h_5 \to h_6,$$

$$h_5 \to h_6 \to h_7 \to h_8 \to h_9 \to h_0 \to h_1.$$

It is not always the case that there are two multiplications that can be paired with each other in an elliptic-curve operation; sometimes there is a need to vectorize a field multiplication internally. We use a similar approach to [3] to compute $h_0, h_1, \ldots, h_9$ in this case; the difference is that we compute vectors $(h_0, h_1), \ldots, (h_8, h_9)$ as result. The strategy for performing the expensive carries on $h_0, h_1, \ldots, h_9$ is the same as [3]. Such an internally-vectorized field multiplication is denoted as $\mathbf{M}$.

2.3 Why Is Smaller Radix Better?

$\mathbf{m}$ takes 29 multiplication instructions (`mul` and `imul`), while $\mathbf{M^2}$ takes $109/2 = 54.5$ multiplication instructions (`vpmuludq`) per field multiplication. How can our software, (which is based on $\mathbf{M^2}$) be faster than [1] (which is based on $\mathbf{m}$) using almost twice as many multiplication instructions?

On Intel microarchitechtures, an instruction is decoded and decomposed into some *micro-operations* (μops). Each μop is then stored in a pool, waiting to be executed by one of the *ports* (when the operands are ready). On each Sandy Bridge and Ivy Bridge core there are 6 ports. In particular, Port 0, 1, 5 are responsible for arithmetic. The remaining ports are responsible for memory access, which is beyond the scope of this paper.

The arithmetic ports are not identical. For example, `vpmuludq` is decomposed into 1 μop, which is handled by Port 0 each cycle with latency 5. `vpaddq` is decomposed into 1 μop, which is handled by Port 1 or 5 each cycle with latency 1. Therefore, an $\mathbf{M^{2-}}$ would take at least

instruction	port	throughput	latency
`vpmuludq`	0	1	5
`vpaddq`	either 1 or 5	2	1
`vpsubq`	either 1 or 5	2	1
`mul`	0 and 1	1	3
`imul`	1	1	3
`add`	either 0, 1, or 5	3	1
`adc`	either two of 0,1,5	1	2

Table 2. Instructions field arithmetic used in [1] and this paper. The data is mainly based on the well-known survey by Fog [10]. The survey does not specify the port utilization for `mul`, so we figure this out using the performance counter (accessed using `perf-stat`). Throughputs are per-cycle. Latencies are given in cycles.

109 cycles. Our experiment shows that $\mathbf{M^{2-}}$ takes around 112 Sandy Bridge cycles, which translates to 56 cycles per multiplication.

The situation for $\mathbf{m}$ is more complicated: `mul` is decomposed into 2 μops, which are handled by Port 0 and 1 each cycle with latency 3. `imul` is decomposed into 1 μop, which is handled by Port 1 each cycle with latency 3. `add` is decomposed into 1 μop, which is handled by one of Port 0,1,5 each cycle with latency 1. `adc` is decomposed into 2 μops, which are handled by two of Port 0,1,5 each cycle with latency 2. In total it takes 25 `mul`, 4 `imul`, 20 `add`, and 20 `adc`, accounting for at least $(25 \cdot 2 + 4 + 20 + 20 \cdot 2)/3 = 38$ cycles. Our experiment shows that $\mathbf{m}^-$ takes 52 Sandy Bridge cycles. The `mov` instructions explain a few cycles out of the $52 - 38 = 14$ cycles. Also, the performance counter shows that the core fails to distribute μops equally over the ports.

Of course, by just looking at these cycle counts it seems that $\mathbf{M^2}$ is still a bit slower, but at least we have shown that the serial multiplier is not as powerful as it seems to be. Here are some more arguments in favor of $\mathbf{M^2}$:

- $\mathbf{m}$ spends more cycles on carries than $\mathbf{M^2}$ does: $\mathbf{m}$ takes 68 Sandy Bridge cycles, while $\mathbf{M^2}$ takes 69.5 Sandy Bridge cycles per multiplication.
- The algorithm built upon $\mathbf{M^2}$ might have additions/subtractions. Some speedup can be gained by interleaving the code; see Section 2.5.
- The computation might have some non-field-arithmetic part which can be improved using vector unit; see Section 3.2.

2.4 Importance of Using a Small Constant

For the ease of reduction, the prime fields used in ECC and HECC are often a big power of 2 subtracted by a small constant c. It might seem that as long as c is not too big, the speed of field arithmetic would remain the same. However, in the following example, we show that using the slightly larger $c = 31$ ($2^{255} - 31$ is the large prime before $2^{255} - 19$) might already cause some overhead.

Consider two field elements f, g which are the results of two field multiplications. Because the limbs are reduced, the upper bound of f_0 would be close to 2^{26}, and the upper bound of f_1 would be close to 2^{25}, and so on; the same bounds apply for g. Now suppose we need to compute $(f - g)^2$, which is batched with another squaring to form an $\mathbf{S}^2$. To avoid possible

underflow, we compute the limbs of $h = f - g$ as $h_i = (f_i + 2 \cdot q_i) - g_i$ instead of $h_i = f_i - g_i$, where q_i is the corresponding limb of $2^{255} - 19$. As the result, the upper bound of h_6 is around $3 \cdot 2^{26}$. To perform the squaring $c \cdot h_6^2$ is required. When $c = 19$ we can simply multiply h_6 by 19 using 1 `vpmuludq`, and then multiply the product by h_6 using another `vpmuludq`. Unfortunately the same instructions do not work for $c = 31$, since $31 \cdot h_6$ can take more than 32 bits.

To overcome such problem, an easy solution is to use a smaller radix so that each (reduced) limb takes fewer bits. This method would increase number of limbs and thus increase number of `vpmuludq` required. A better solution is to delay the multiplication by c: instead of computing $31f_{i_1}g_{j_1} + 31f_{i_2}gj_2 + \cdots$ by first computing $31g_{j_1}, 31g_{j_2}, \ldots$, compute $f_{i_1}g_{j_1} + f_{i_2}g_{j_2} + \cdots$ and then multiply the sum by 31. The sum can take more than 32 bits (and `vpmuludq` takes only 32-bit inputs), so the multiplication by 31 cannot be handled by `vpmuludq`. Let $s = f_{i_1}g_{j_1} + f_{i_2}g_{j_2} + \cdots$, one way to handle the multiplication by 19 is to compute $32s$ with one shift instruction `vpsllq` and then compute $32s - s = 31s$ with one subtraction instruction `vpsubq`. This solution does not make Port 0 busier as `vpsllq` also takes only one cycle in Port 0 as `vpmuludq`, but it does make Port 1 and 5 busier (because of `vpsubq`), which can potentially increase the cost for $\mathbf{S^{2-}}$ by a few cycles.

It is easy to imagine for some c's the multiplication can not be handled in such a cheap way as 31. In addition, delaying multiplication cannot handle as many c's as using a smaller radix; as a trivial example, it does not work if $cf_{i_1}g_{j_1} + cf_{i_2}g_{j_2} + \cdots$ takes more than 64 bits. We note that the computation pattern in the example is actually a part of elliptic-curve operation (see lines 6–9 in Algorithm 1), meaning a bigger constant c actually can slow down elliptic-curve operations.

We comment that usage of a larger c has bigger impact on constrained devices. If c is too big for efficient vectorization, at least one can go for the $64 \times 64 \rightarrow 128$-bit serial multiplier, which can handle a wide range of c without increasing number of limbs. However, on ARM processors where the serial multiplier can only perform $32 \times 32 \rightarrow 64$-bit multiplications, even the serial multiplier would be sensitive to the size of c. For even smaller devices the situation is expected to be worse.

2.5 Instruction Scheduling for Vectorized Field Arithmetic

The fact that μops are stored in a pool before being handled by a port allows the CPU to achieve so called *out-of-order execution*: a μop can be executed before another μop which is from an earlier instruction. This feature is sometimes viewed as the CPU core being able to "look ahead" and execute a later instruction whose operands are ready. However, the ability of out-of-order execution is limited: the core is not able to look too far away. It is thus better to arrange the code so that each code block contains instructions for each port.

While Port 0 is quite busy in $\mathbf{M^2}$, Port 1 and 5 are often idle. In an elliptic-curve operation (see the following sections) an $\mathbf{M^2}$ is often preceded by a few field additions/subtractions. Since `vpaddq` and the vectorized subtraction instruction `vpsubq` can only be handled by either Port 1 and Port 5, we try to interleave the multiplication code with the addition/subtraction code to reduce the chance of having an idle port. Experiment results show that the optimization brings a small yet visible speedup. It seems more difficult for an algorithm built upon $\mathbf{m}$ to use the same optimization.

3 The Curve25519 elliptic-curve-Diffie-Hellman scheme

[11] defines Curve25519 as a function that maps two 32-byte input strings to a 32-byte output string. The function can be viewed as an x-coordinate-only scalar multiplication on the curve

$$E_M : y^2 = x^3 + 486662x^2 + x$$

over $\mathbb{F}_{2^{255}-19}$. The curve points are denoted as $E_M(\mathbb{F}_{2^{255}-19})$. The first input string is interpreted as an integer scalar s, while the second input string is interpreted as a 32-byte encoding of x_P, the x-coordinate of a point $P \in E_M(\mathbb{F}_{2^{255}-19})$; the output is the 32-byte encoding of x_{sP}.

Given a 32-byte secret key and the 32-byte encoding of a standard base point defined in [11], the function outputs the corresponding public key. Similarly, given a 32-byte secret key and a 32-byte public key, the function outputs the corresponding shared secret. Although the same routine can be used for generating both public keys and shared secrets, the public-key generation can be done much faster by performing the scalar multiplication on an equivalent curve. The rest of this section describes how we implement the Curve25519 function for shared-secret computation and public-key generation.

3.1 Shared-Secret Computation

Algorithm 1 The Montgomery ladder step for Curve25519

function LADDERSTEP(x_2, z_2, x_3, z_3, x_P)
  $t_0 \leftarrow x_3 - z_3$
  $t_1 \leftarrow x_2 - z_2$
  $x_2 \leftarrow x_2 + z_2$
  $z_2 \leftarrow x_3 + z_3$
  $z_3 \leftarrow t_0 \cdot x_2$; $z_2 \leftarrow z_2 \cdot t_1$ ▷ batched multiplications
  $x_3 \leftarrow z_3 + z_2$
  $z_2 \leftarrow z_3 - z_2$
  $x_3 \leftarrow x_3^2$; $z_2 \leftarrow z_2^2$ ▷ batched squarings
  $z_3 \leftarrow x_P \cdot z_2$;
  $t_0 \leftarrow t_1^2$; $t_1 \leftarrow x_2^2$ ▷ batched squarings
  $x_2 \leftarrow t_1 - t_0$
  $z_3 \leftarrow x_2 \cdot 121666$
  $z_2 \leftarrow t_0 + z_3$
  $z_2 \leftarrow x_2 \cdot z_2$; $x_2 \leftarrow t_1 \cdot t_0$ ▷ batched multiplications
  return (x_2, z_2, x_3, z_3)
end function

The best known algorithm for x-coordinate-only variable-base-point scalar multiplication on Montgomery curves is the *Montgomery ladder*. [1], [3] and our software all use the Montgomery ladder for Curve25519 shared secret computation. Similar to the double-and-add algorithm, the algorithm also iterates through each bit of the scalar, from the most significant to the least significant one. For each bit of the scalar the ladder performs a differential addition and a doubling. The differential addition and the doubling together are called a *ladder step*. Since the ladder step can be carried out by a fixed sequence of field operations, the Montgomery ladder is almost intrinsically constant-time. We summarize the ladder step for Curve25519 in Algorithm 1. Note that Montgomery uses projective coordinates.

In order to make the best use of the vector unit (see Section 2), multiplications and squarings are handled in pairs whenever convenient. The way we pair multiplications is shown in the comments of Algorithm 1. It is not specified in [3] whether they pair multiplications and squarings in the same way, but this seems to be the most natural way. Note that the multiplication by 121666 (line 13) and the multiplication by x_1 (line 10) are not paired with other multiplications. We deal with the two multiplications as follows:

- Compute multiplications by 121666 without carries using 5 `vpmuludq`.
- Compute multiplications by x_1 without carries. This can be completed in 50 `vpmuludq` since we precompute the products of small constants (namely, 2, 19, and 38) and limbs in x_1 before the ladder begins.
- Perform batched carries for the two multiplications.

This uses far fewer cycles than handling the carries for the two multiplications separately.

Note that we often have to "transpose" data in the ladder step. More specifically, after an $\mathbf{M^2}$ which computes $(h_0, h'_0), \ldots, (h_9, h'_9)$, we might need to compute $h+h'$ and $h-h'$; see lines 6–8 of Algorithm 1. In this case, we compute (h_i, h_{i+1}), (h'_i, h'_{i+1}) from $(h_i, h'_i), (h_{i+1}, h'_{i+1})$ for $i \in \{0, 2, 4, 6, 8\}$, and then perform additions and subtractions on the vectors. The transpositions can be carried out using the "unpack" instructions `vpunpcklqdq` and `vpunpckhqdq`. Similarly, to obtain the operands for $\mathbf{M^2}$ some transpositions are also required. Unpack instructions are the same as `vpaddq` and `vpsubq` in terms of port utilization, so we also try to interleave them with $\mathbf{M^2}$ or $\mathbf{S^2}$ as described in Section 2.5.

3.2 Public-Key Generation

Instead of performing a fixed-base scalar multiplication directly on the Montgomery curve, we follow [7] to perform a fixed-base scalar multiplication on the twisted Edwards curve

$$E_T : -x^2 + y^2 = 1 - 121665/121666x^2y^2$$

over $\mathbb{F}_{2^{255}-19}$ and convert the result back to the Mongomery curve with one inversion. The curve points are denoted as $E_T(\mathbb{F}_{2^{255}-19})$. There is an efficiently computable birational equivalence between E_T and E_M, which means the curves share the same group structure and ECDLP difficulty. Unlike Mongomery curves, there are complete formulas for point addition and doubling on twisted Edwards curves; we follow [1] to use the formulas for the extended coordinates proposed in [9]. The complete formulas allow utilization of a table of many precomputed points to accelerate the scalar multiplication, which is the reason fixed-base multiplications (on both E_M and E_T) can be carried out much faster than variable-base scalar multiplications.

In [1] a fixed-base scalar multiplication sB where $s \in \mathbb{Z}$ and $B \in E_T(\mathbb{F}_{2^{255}-19})$ (B corresponds to the standard base point in $E_M(\mathbb{F}_{2^{255}-19})$) is performed as follows: write s (modulo the order of B) as $\sum_{i=0}^{15} 16^i s_i$ where $s_i \in \{-8, -7, \ldots, 7\}$ and obtain sB by computing the summation of $s_0B, s_1 16B, \ldots, s_{15}16^{15}B$. To obtain $s_i 16^i B$, the strategy is to precompute several multiples of $16^i B$ and store them in a table, and then perform a constant-time table lookup using s_i as index on demand. [1] also shows how to reduce the size of the table by dividing the sum into two parts:

$$P_0 = s_0B + s_2 16^2 B + \cdots + s_{14}16^{14}B$$

and

$$P_1 = s_1B + s_3 16^2 B + \cdots + s_{15} 16^{14} B.$$

$sB = P_0 + 16P_1$ is then obtained with 4 point doublings and 1 point addition. In this way, the table contains only multiples of $B, 16^2B, \ldots, 16^{14}B$.

We do better by vectorizing between computations of P_0 and P_1: all the data related to P_0 and P_1 are loaded into the lower half and upper half of the 128-bit registers, respectively. This type of computation pattern is very friendly for vectorization since there no need to "transpose" the data as in the case of Section 3.1.

While parallel point additions can be carried out easily, an important issue is how to perform parallel constant-time table lookups in an efficient way. In [1], suppose there is a need to lookup s_iP, the strategy is to precompute a table containing $P, 2P, \ldots, 8P$, and then the lookup is carried out in two steps:

- Load $|s_i|P$ in constant time, which is the main bottleneck of the table lookup.
- Negate $|s_i|P$ if s_i is negative.

For the first step it is convenient to use the conditional move instruction (`cmov`): To obtain each limb (of each coordinate) of $|s_i|P$, first initialize a 64-bit register to the corresponding limb of ∞, then for each of $P, 2P, \ldots, 8P$, conditionally move the corresponding limb into the register. Computation of the conditions and conditional negation are relatively cheap compared to the `cmov` instructions. [1] uses a 3-coordinate system for precomputed points, so the table-lookup function takes $3 \cdot 8 \cdot 5 = 120$ `cmov` instructions. The function takes 159 Sandy Bridge cycles or 158 Ivy Bridge cycles.

We could use the same routine twice for parallel table lookups, but we do better by using vector instructions. Here is a part of the inner loop of our qhasm ([18]) code.

```
v0 = mem64[input_1 + 0] x2
v1 = mem64[input_1 + 40] x2
v2 = mem64[input_1 + 80] x2
v0 &= mask1
v1 &= mask1
v2 &= mask1
t0 |= v0
t1 |= v1
t2 |= v2
```

The first line `v0 = mem64[input_1 + 0] x2` loads the first and second limb (each taking 32 bits) of the first coordinate of P and broadcasts the value to the lower half and upper half of the 128-bit register `v0` using the `movddup` instruction. The line `v0 &= mask1` performs a bitwise AND of `v0` and a mask; the value in the mask depends on whether $s_i = 1$. Finally, `v0` is ORed into `t0`, which is initialized in a similar way as in the `cmov`-based approach. Similarly, the rest of the lines are for the second and third coordinates. Similar code blocks are repeated 7 more times for $2P, 3P, \ldots, 8P$, and all the code blocks are surrounded by a loop which iterates through all the limbs. In total it takes $3 \cdot 8 \cdot 5 \cdot 2 = 240$ logic instructions. The parallel table-lookup function (inlined in our implementation) takes less than 160 Sandy/Ivy Bridge cycles, which is almost twice as fast as the `cmov`-based table lookup function.

Algorithm 2 The doubling function for twisted Edwards curves

1: **function** GE_DBL_P2(X, Y, Z)
2: $A \leftarrow X^2; B \leftarrow Y^2$ ▷ batched squarings
3: $G \leftarrow A - B$
4: $H \leftarrow A + B$
5: $C \leftarrow 2Z^2; D = (X + Y)^2$ ▷ batched squarings
6: $E \leftarrow H - D$
7: $I \leftarrow G + C$
8: $X' \leftarrow E \cdot I; Y' \leftarrow G \cdot H$ ▷ batched multiplications
9: $Z' \leftarrow G \cdot I$
10: **return** (X', Y', Z')
11: **end function**

4 Vectorizing the Ed25519 signature scheme

This section describes how the Ed25519 verification is implemented with focus on the challenge of vectorization. Since the public-key generation and signing process, as the Curve25519 public-key generation, is bottlenecked by a fixed-base scalar multiplication on E_T, the reader can check Section 3.2 for the implementation strategy.

4.1 Ed25519 Verification

[1] verifies a message by computing the double-scalar multiplication of the form $s_1P_1 + s_2P_2$. The double-scalar multiplication is implemented using a generalization of the sliding-window method such that s_1P_1 and s_2P_2 share the doublings. With the same window sizes, we do better by vectorizing the point doubling and point addition functions.

On average each verification takes about 252 point doublings, accounting for more than 110000 cycles. There are two doubling functions in our implementation; `ge_dbl_p2`, which is adapted from the "$\mathcal{E} \leftarrow 2\mathcal{E}$" doubling described in [9], is the most frequently used one; see [9] for the reason to use different doubling and addition functions. On average `ge_dbl_p2` is called 182 times per verification, accounting for more than 74000 cycles. The function is summarized in Algorithm 2. Given $(X : Y : Z)$ representing $(X/Z, Y/Z) \in E_T$, the function returns $(X' : Y' : Z') = (X : Y : Z) + (X : Y : Z)$. As in Section 3.1, squarings and multiplications are paired whenever convenient. However it is not always possible to do so, as the multiplication in line 9 can not be paired with other operations. The single multiplication slows down the function, and the same problem also appears in addition functions.

Another problem is harder to see. $E = X^2 + Y^2 - (X + Y)^2$ has limbs with upper bound around $4 \cdot 2^{26}$, and $I = X^2 - Y^2 + 2Z^2$ has limbs with upper bound around $5 \cdot 2^{26}$. For the multiplication $E \cdot I$, limbs of either E or I have to be multiplied by 19 (see Section 2.2), which can be more than 32 bits. This problem is solved by performing extra carries on limbs in E before the multiplication. The same problem appears in the other doubling function.

In general the computation pattern for verification is not so friendly for vectorization. However, even in this case our software still gains non-negligible speedup over [1] and [7]. We conclude that the power of vector unit on recent Intel microarchitectures might have been seriously underestimated, and implementors for ECC software should consider trying vectorized multipliers instead of serial multipliers.

References

[1] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, Bo-Yin Yang, *High-speed high-security signatures*, in CHES 2011 (2011), 124-142. Citations in this document: §1, §1.1, §1.2, §1.2, §1.2, §1.2, §1.2, §1, §1, §1.2, §1.2, §1.2, §1.2, §1.2, §1.2, §1.2, §1.3, §2, §2, §2.1, §1, §2, §2, §2.3, §3.1, §3.2, §3.2, §3.2, §3.2, §3.2, §4.1, §4.1.

[2] Neil Costigan, Peter Schwabe, *Fast elliptic-curve cryptography on the Cell Broadband Engine*, in AFRICACRYPT 2009 (2009), 368–385. Citations in this document: §1, §1.1.

[3] Daniel J. Bernstein, Peter Schwabe, *NEON crypto*, in CHES 2012 (2012), 320–339. Citations in this document: §1, §1.1, §1.1, §2, §2.2, §2.2, §2.2, §2.2, §3.1, §3.1.

[4] Daniel J. Bernstein, Chitchanok Chuengsatiansup, Tanja Lange, Peter Schwabe, *Kummer strikes back: new DH speed records*, in EUROCRYPT 2015 (2014), 317–337. Citations in this document: §1.1, §1.1, §1.3, §1.3, §1.3.

[5] Craig Costello, Huseyin Hisil, Benjamin Smith, *Faster compact Diffie–Hellman: endomorphisms on the x-line*, in EUROCRYPT 2014 (2014), 183–200. Citations in this document: §1.1.

[6] Patrick Longa, Francesco Sica, Benjamin Smith, *Four-dimensional Gallant–Lambert–Vanstone scalar multiplication*, in Asiacrypt 2012 (2012), 718–739. Citations in this document: §1.1.

[7] Andrew M. "Floodyberry", *Implementations of a fast Elliptic-curve Digital Signature Algorithm* (2013). URL: https://github.com/floodyberry/ed25519-donna. Citations in this document: §1.2, §1.2, §1.2, §1.2, §1, §1, §1.2, §1.2, §1.2, §1.2, §1.2, §1.2, §3.2, §4.1.

[8] Daniel J. Bernstein, Tanja Lange (editors), *eBACS: ECRYPT Benchmarking of Cryptographic Systems* (2014). URL: http://bench.cr.yp.to. Citations in this document: §1, §1, §1, §1, §1.2, §1.3.

[9] Huseyin Hisil, Kenneth Koon-Ho Wong, Gary Carter, Ed Dawson, *Twisted Edwards curves revisited*, in Asiacrypt 2008 (2008), 326–343. Citations in this document: §3.2, §4.1, §4.1.

[10] Agner Fog, *Instruction tables* (2014). URL: http://www.agner.org/optimize/instruction_tables.pdf. Citations in this document: §2, §2.

[11] Daniel J. Bernstein, *Curve25519: new Diffie-Hellman speed records*, in PKC 2006 (2006), 207–228. Citations in this document: §3, §3.

[12] Thomaz Oliveira, Julio López, Diego F Aranha, Francisco Rodríguez-Henríquez,, *Lambda coordinates for binary elliptic curves*, in CHES 2013 (2013), 211–330. Citations in this document: §1.3.

[13] Joppe W Bos, Craig Costello, Huseyin Hisil, Kristin Lauter, *Fast cryptography in genus 2*, Journal of Cryptology (2013), 1–33. Citations in this document: §1.3.

[14] Christophe Petit, Jean-Jacques Quisquater, *On polynomial systems arising from a Weil descent*, in ASIACRYPT 2012 (2012), 451–466. Citations in this document: §1.3.

[15] Igor Semaev, *New algorithm for the discrete logarithm problem on elliptic curves* (2015). URL: https://eprint.iacr.org/2015/310.pdf. Citations in this document: §1.3.

[16] Michael Düll, Björn Haase, Gesine Hinterwälder, Michael Hutter, Christof Paar, Ana Helena Sánchez, Peter Schwabe, *High-speed Curve25519 on 8-bit, 16-bit and 32-bit microcontrollers*, Design, Codes and Cryptography (to appear). URL: http://cryptojedi.org/papers/#mu25519. Citations in this document: §1.

[17] — (no editor), *IANIX*. URL: ianix.com. Citations in this document: §1.

[18] Daniel J. Bernstein, *qhasm sofware package* (2007). URL: http://cr.yp.to/qhasm.html. Citations in this document: §3.2.

[19] Patrick Longa, *NUMS Elliptic Curves and their Implementation*. URL: http://patricklonga.webs.com/NUMS_Elliptic_Curves_and_their_Implementation-UoWashington.pdf. Citations in this document: §1.2.

[20] Nicolas Thériault, *Index calculus attack for hyperelliptic curves of small genus*, in Asiacrypt 2003 (2003), 75–92. Citations in this document: §1.3.